Static Analysis for Software Assurance: Soundness, Scalability and Adaptiveness

Arnaud J. Venet
CMU / NASA Ames Research Center
Moffett Field, CA 94035
arnaud.j.venet@nasa.gov

Michael R. Lowry
NASA Ames Research Center
Moffett Field, CA 94035
michael.r.lowry@nasa.gov

ABSTRACT

Standard approaches to software assurance are either process-based or test-based. We propose to include static analysis by Abstract Interpretation to the software development cycle. Static analysis by Abstract Interpretation provides a high level of assurance as well as ground-truth evidence in support of its findings. Successes in the verification of large industrial codes demonstrate the readiness of this technology. However, in order to be practical in real development environments, static analysis must be able to scale and yield few false positives without the need for expert hand-tuning. We present a research agenda to reach this goal based on the development of adaptive static analysis algorithms.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Theory, Verification, Measurement

Keywords

Abstract Interpretation, Software Certification, Static Analysis

1. VISION

Automated software verification tools that provide guarantees can dramatically change the process and economics of developing certifiable software systems. Without guarantees, a verification tool is only *advisory*, and cannot substitute for any human assurance activity such as those prescribed under DO178B. However, if a tool can provide both guarantees of finding certain classes of defects (no false negatives) and sufficient precision to minimize false positives so filtering them is economical, then such a tool can become an integral part of the certification process. As the range of defects that can be detected in this manner is expanded, as well as the size of systems that can be verified, the process of certification will become increasingly automated.

The three objectives of no false negatives, high precision (few false positives), and scaling to large systems push the boundaries of computational complexity. In other words, except for especially simple classes of defects, no single verification algorithm will be able to achieve all three. However, in case studies described here and elsewhere, human experts have demonstrated that given a particular software system being certified; they can adapt a custom algorithmic approach by carefully selecting, composing, and tuning known verification algorithms to simultaneously achieve these three objectives.

Can this adaptive expert human approach of selecting and composing verification algorithms be automated? In this position paper we discuss this approach within the context of Abstract Interpretation for static analysis. Then we provide evidence that automated adaptation to a given problem can achieve all three objectives. We also describe an agenda for generalizing the approach beyond static analysis of software.

Static analysis is increasingly used to look for hidden defects in industrial software. The commercialization of tools like Coverity Prevent [9] or CodeSonar [12], which have been shown to scale to large code bases and find real bugs, helped popularize static analysis technology among software developers. However, in terms of software assurance this technology does not provide *ground truth*: all defect reports are mere warnings, which may turn out to be spurious (false positives), whereas real defects may go undetected (false negatives). This class of static analyzers do not produce any strong evidence in support of the defects reported nor do they provide any metrics for calibrating false negatives.

A theory devised more than 30 years ago and named Abstract Interpretation [6] enables the construction of static analyzers that do not yield any false negatives. Abstract Interpretation provides a methodology to mathematically derive an algorithm for computing a given class of program properties from a formal definition of the semantics of the programming language considered. The static analyzer obtained in this way is not *complete*, in the sense that it cannot identify all defects of a certain category for all programs with absolute certainty, which translates into false positives being produced. However, the analyzer is *sound* i.e., all program defects that fall within the scope of the analyzer are detected. Hence there are no false negatives within the scope.

Long considered to be impractical for analyzing real-life code, in recent years Abstract Interpretation has been successfully applied to the verification of large aerospace applications [24, 7, 3, 2]. Customization is the key for achieving both scalability and low false positive rate: the abstraction interpretation algorithms are tailored for a special code or family of codes using knowledge of the application domain (software architecture, use of certain numerical algorithms, types of data structures manipulated, etc.). We consider this evolution as a major step toward the use of static analysis as a process that can automatically assert ground truth on some important classes of software properties.

In this position paper, we advocate for the use of static analysis by abstract interpretation as a fully automated certification process for modern software assurance. Abstract interpretation bridges a gap that standard software assurance techniques cannot address: certification standards like DO-178B are solely concerned with the development process, testing can only cover so many of all possible behaviors of the application in the field, and common static analysis tools do not provide evidence of the absence of defects.

The paper is organized as follows: Section 2 describes the current state of practice for static analysis based on Abstract Interpretation and highlights the limits of existing implementations. In Sect. 3 we outline research directions that would make these static analyzers more adaptable to a variety of industrial development environments. In Sect. 4 we propose to apply Abstract Interpretation to other phases of the development cycle and use it in combination with different verification technologies.

2. CURRENT STATE OF PRACTICE FOR ABSTRACT INTERPRETATION

Abstract Interpretation [6] breaks down the static analysis of a program into a number of abstractions and operations that are formally stated and can be mathematically proven to correctly represent the semantics of the programming language considered (what is usually called *soundness*). Probably the single most important structure in Abstract Interpretation is the *abstract domain*. An abstract domain is used to represent sets of memory configurations which make up *program invariants*. For example, the abstract domain of intervals represents all possible values of scalar variables at a certain point in the program by their ranges. The domain of convex polyhedra [8] is more expressive than intervals since it can represent linear inequality constraints among program variables. Other abstract domains may be used to represent floating-point values [5] or pointers [23]. A static analyzer combines several abstract domains in order to compute an abstraction of all possible types of data manipulated by the program.

A static analyzer based on abstract interpretation does not look for a certain category of defects, but rather computes an abstraction of all possible memory configurations at each point in the program using abstract domains. Program invariants can be readily extracted from abstract domains and then used to prove or disprove the safety of certain operations in the program. For example, the abstract domain of intervals can be used to check each arithmetic operation in the program for possible overflows or underflows. The information provided by abstract domains can be displayed to the user as ground-truth evidence of the formal verification process and used to independently verify the validity of the static analysis results. This comes in sharp contrast with commercial bug-finding static analyzers, which provide limited feedback on example executions that expose a problem.

However, with current technology, it is not possible to build a static analyzer that is both accurate and efficient. Precise abstract domains come with hefty computational costs. The domain of intervals is efficient but not precise enough in some situations. The domain of convex polyhedra is very precise but its computational complexity is so high that it cannot be reasonably applied over more than fifteen variables in practice. There is no one universal abstract domain (or combination of abstract domains) that is computationally tractable and provides high precision for static analysis of all programs.

The first generation of static analyzers based on Abstract Interpretation, like PolySpace [21], had built-in precision levels, that each implemented a certain precision vs. speed trade-off and that could be selected by the user. As a result, only mid-size programs could be analyzed and a high number of false positives were generally produced. The main observation is that abstract domains should not be used *uniformly* over the whole application, but in a more local manner, powerful algorithms being used on small portions of the code whenever extra precision is needed.

This observation has led to the development of a second generation of static analyzers, like C Global Surveyor [24] and ASTREE [7], that could both scale to large codes and achieve low false positive rate. Among the techniques used was the application of precise numerical domains over small packets of variables and the design of special-purpose abstract domains for the analysis of certain algorithms, like linear digital filters [11]. ASTREE could successfully analyze the fly-by-wire software of the Airbus A380 (over 400K LOC), while C Global Surveyor could be applied to the flight software of the NASA Mars Exploration Rovers mission (over 550K LOC).

Those kind of static analyzers are able to scale to large codes but can only verify simple program properties, like the absence of runtime errors (out-of-bounds array access, null pointer dereference, arithmetic overflow, etc.). Three-Valued Logic [15] is a static analysis framework based on Abstract Interpretation that allows the verification of more advanced program properties, in particular for dynamic data structures [20], like the correctness of rebalancing operations for AVL trees [19]. However, these techniques are computationally more demanding and can only be applied to smaller software components.

The downside of the Abstract Interpretation approach is that it requires considerable hand-tuning by experts, who must also acquire an intimate knowledge of the application analyzed. In the case of ASTREE for example, a special family of abstract domains had to be designed in order to precisely analyze numerical algorithms implementing digital linear filters [11]. When analyzing a program with Three-Valued Logic, it is necessary to introduce instrumentation predicates, which are auxiliary properties that are necessary for the analyzer to prove the desired property on a specific program [18]. These are complex steps that cannot be realistically performed by someone who is just a user of the tool. Moreover, such a custom analyzer can only work for one application or a very particular family of applications. It is not realistic to advocate for a broad use of static analysis as a certification technique in the software development process if it requires so much expert work.

3. TOWARD ADAPTIVE STATIC ANALYZERS

A central technique to achieve scalability in both ASTREE and C Global Surveyor is the variable packing technique mentioned above. ASTREE performs the packing of variables statically, based on the variables that occur in one statement, and there is no interaction among overlapping variable packets at analysis time. These restrictive assumptions were acceptable because of the particular nature of the code analyzed. In contrast, C Global Surveyor performs variable packing on-the-fly during the analysis. Two variables are grouped together only if there is a computational dependence between them, and packets can be merged as the analysis proceeds. This abstract domain is *adaptive* and is not restricted to a particular class of software.

C Global Surveyor has been initially designed to analyze a certain architecture of flight code shared by several NASA missions, specifically Mars Path Finder, Deep Space 1, and Mars Exploration Rovers. When C Global Surveyor was applied to different codes from the Space Shuttle or the International Space Station, it achieved similar levels of speed and precision [3]. This consistent behavior is essentially due to the adaptive abstract domain, which groups variables according to a purely *semantic* criterion as opposed to the syntactic packing performed by ASTREE.

The adaptive abstract domain implemented C Global Surveyor is complex. It essentially consists of an evolving abstract domain, the structure of which may change numerous times during the analysis. There is a general mathematical construction that allows the step-by-step construction of such domains, characterized as *cofibered domains* [22], from simpler existing abstract domains. Cofibered domains can be used as a building block to design other kinds of abstract domains that cover different aspects of the analysis of an application. Whenever a specialized abstract domain is required to precisely analyze certain parts of an application, instead of making a one-shot specialization of the analyzer, we propose to make the abstract domain adaptive. Therefore, the newly introduced abstract domain can automatically transpose to different applications, without the need for hand-tuning whenever a new code is analyzed.

In this new formulation for adaptive static analysis, we advocate for a flexible adaptation of the analysis algorithms to the problem structure, based on continuous improvement in the problem structure understanding. This approach enriches the capabilities of the analyzer without tying it up to a particular kind of software. By removing the need for manual tuning of the analyzer, we believe that this approach would lead to static analyzers that can be used in broader development environments. In the rest of this section, we will lay out the main components of a research agenda along these lines.

Without delving into technical details, we can just say that Abstract Interpretation is based on the order-theoretic notion of a lattice [10]. Given a formal definition of the semantics of the program, the powerset of all possible program configurations is a lattice $\mathbb{D}$ and the execution of the program can be modeled as the least fixpoint of a function $\mathbb{F} : \mathbb{D} \to \mathbb{D}$. The main idea of Abstract Interpretation is to come up with a lattice D, called an *abstract domain*, and a function $F : D \to D$ that together form a sound approximation of the semantics of the program i.e., there exists a morphism $\gamma : D \to \mathbb{D}$ such that $\mathbb{F} \circ \gamma \sqsubseteq \gamma \circ F$. The main activity in Abstract Interpretation is to come up with a machine representable abstract domain D and a computable function F, the fixpoint of which can be computed in finite time. The precision and performance of the resulting static analyzer are almost entirely determined by the choice of the abstract domain. Popular abstract domains include the polyhedral domain [8], the linear equality domain [14] and the octagon domain [17].

Making Abstract Interpretation adaptive implies considering a family of abstract domains $(D_i)_{i \in I}$ instead of a single one. The abstract domain changes at analysis time in order to adapt to various situations. The analysis must be able to smoothly transfer from one domain to the other without any loss of information, so as to preserve soundness. Cofibered domains [22] are one such attempt to formalize this situation in a general Abstract Interpretation setting. Practical approaches to refine the domain of abstraction include using inductive inference for Three-Valued Logic [16] or lazy abstraction [1, 13] for Boolean models.

Another important research direction is the study of the connection between a specialized abstract domain and the code on which it applies. Specialized abstract domains, like the domain of invariants for linear digital filters [11], need only be activated when a relevant piece of code is analyzed. Currently, the static analyzer is hand-tuned and/or uses syntactic pattern-matching algorithms to decide when to enable the abstract domain and when to disable it. In order to be adopted by non-expert users, the static analyzer must be able to automatically infer when to trigger a specific abstract domain. This implies being able to recognize a certain code structure or algorithm, based no longer on the syntax of the program but on purely semantic grounds. This means designing a static analysis that is able to infer a semantic signature of a certain class of algorithms.

More generally, a fully adaptive static analysis framework would allow, for example, the use of a shape analysis algorithm on one part of the program, a lazy-abstraction based analysis on another part and a polyhedral invariant analysis on loop nests. This requires automatically shifting not only abstractions but also classes of abstractions, e.g., from symbolic to Boolean and then to numerical, smoothly and in a sound manner. To the best of our awareness, this topic has not received much attention in the literature on static analysis, but would prove very important for the development of fully automated adaptive static analyzers.

4. BEYOND CODE ANALYSIS

Abstract Interpretation can be defined as a theory of discrete approximation. It is essentially used to design sound static analyzers but this is not the only field of application of the theory. Abstract domains that provide computable representations of sets of numerical values are interesting in their own for the analysis of discrete systems in general, not just code. For example, the technique of *abstract simulation* has been developed to estimate round-off errors introduced by numerical algorithms modeled in Simulink [4]. Abstract simulation is performed directly on the Simulink diagram, which means that one is able to verify numerical behaviors of embedded systems at the design level.

This is significant, since the cost of fixing a design error gets dramatically higher when it is detected later in the development cycle. More generally, block diagram specifications can be subject to analysis by Abstract Interpretation as long as they can be endowed with a formal semantics. This is particularly interesting for UML specifications, since this would allow the verification of properties of the system early in the development cycle, when no code is present.

At the other end of the verification process, the program invariants generated by a static analyzer can be used to enhance testing. For example, inferred ranges for variables may be employed to narrow down the search for test input data, whereas interval bounds may reveal hidden edge cases. This information is also valuable to an explicit-state model checker as it may substantially cut down the search space. The synergistic combination of model checking and static analysis looks promising, as model checkers are good at finding issues, like deadlocks, that are difficult to detect using static analysis only.

As a conclusion, we would like to propose the use of Abstract Interpretation techniques at all stages of the software development process in support of and/or in combination with other software assurance approaches.

5. REFERENCES

[1] T. Ball and S. K. Rajamani. The slam project: debugging system software via static analysis. In *POPL*, pages 1–3, 2002.

[2] O. Bouissou, E. Conquet, P. Cousot, R. Cousot, J. Feret, K. Ghorbal, É. Goubault, D. Lesens, L. Mauborgne, A. Miné, S. Putot, X. Rival, and M. Turin. Space software validation using abstract interpretation. In *Proc. of the Int. Space System Engineering Conf., Data Systems in Aerospace (DASIA 2009)*, volume SP-669, pages 1–7, May 2009.

[3] G. Brat and Arnaud Venet. Precise and scalable static program analysis of NASA flight software. In *Proceedings of the IEEE Aerospace Conference*, Big Sky, MT, 2005.

[4] A. Chapoutot and M. Martel. Abstract simulation: a static analysis of simulink models. In *6th IEEE International Conference on Embedded Systems and Software (ICESS'09)*, 2009.

[5] L. Chen, A. Miné, J. Wang, and P. Cousot. A sound floating-point polyhedra abstract domain. In G. Ramalingam, editor, *Proceedings of the sixth Asian Symposium (APLAS'08)*, pages 3–18, Bangalore, India, 2009.

[6] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[7] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE Analyser. In M. Sagiv, editor, *Proc. of the European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30, April 2–10 2005.

[8] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978.

[9] Coverity. Prevent. (`www.coverity.com`).

[10] B. A. Davey and H. A. Priestley. *Introduction to lattices and order*. Cambridge University Press, Cambridge, 1990.

[11] J. Féret. The arithmetic-geometric progression abstract domain. In *Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, number 3385 in LNCS, pages 42–58, 2005.

[12] GrammaTech. CodeSonar. (`www.grammatech.com`).

[13] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.

[14] M. Karr. Affine relationships among variables of a program. *Acta Inf.*, 6:133–151, 1976.

[15] T. Lev-Ami and S. Sagiv. Tvla: A system for implementing static analyses. In *Static Analysis Symposium*, pages 280–301, 2000.

[16] A. Loginov, T. W. Reps, and S. Sagiv. Abstraction refinement via inductive learning. In *CAV*, pages 519–533, 2005.

[17] A. Miné. The octagon abstract domain. In *Proc. of the Workshop on Analysis, Slicing, and Transformation (AST'01)*, IEEE, pages 310–319, October 2001.

[18] T. W. Reps, S. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. In *VMCAI*, pages 252–266, 2004.

[19] R. Rugina. Quantitative shape analysis. In *Static Analysis Symposium*, pages 228–245, 2004.

[20] S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[21] The MathWorks. PolySpace. (`http://www.mathworks.com/products/polyspace`).

[22] A. Venet. Abstract cofibered domains: Application to the alias analysis of untyped programs. In *SAS*, pages 366–382, 1996.

[23] A. Venet. A scalable nonuniform pointer analysis for embedded programs. In *SAS*, pages 149–164, 2004.

[24] A. Venet and G. P. Brat. Precise and efficient static array bound checking for large embedded C programs. In *Proc. of the Int. Conf. on Programming Language Design and Implementation (PLDI)*, pages 231–242, june 2004.